January 2002

Validating Constraints in XML

Yi Chen
University of Pennsylvania

Susan B. Davidson
University of Pennsylvania, susan@cis.upenn.edu

Yifeng Zheng
University of Pennsylvania

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-02-03.

Validating Constraints in XML

Abstract

The role of XML in data exchange is evolving from one of merely conveying the structure of data to one that also conveys its semantics. In particular, several proposals for key and foreign key constraints have recently appeared, and aspects of these proposals have been adopted within XMLSchema. Although several validators for XMLSchema appear to check for keys, relatively little attention has been paid to the general problem of how to check constraints in XML.

In this paper, we examine the problem of checking keys in XML documents and describe a native validator based on SAX. The algorithm relies on an indexing technique based on the paths found in key definitions, and can be used for checking the correctness of an entire document (bulk checking) as well as for checking updates as they are made to the document (incremental checking). The asymptotic performance of the algorithm is linear in the size of the document or update. We also discuss how XML keys can be checked in relational representations of XMLdocuments, and compare the performance of our native validator against hand-coded relational constraints. Extrapolating from this experience, we propose how a relational schema can be designed to check XMLSchema key constraints using efficient relational PRIMARY KEY or UNIQUE constraints.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-02-03.

Validating Constraints in XML

Yi Chen, Susan B. Davidson and Yifeng Zheng
Dept. of Computer and Information Science
University of Pennsylvania
yicn@saul.cis.upenn.edu, susan@cis.upenn.edu, yifeng@saul.cis.upenn.edu

Abstract

The role of XML in data exchange is evolving from one of merely conveying the structure of data to one that also conveys its semantics. In particular, several proposals for key and foreign key constraints have recently appeared, and aspects of these proposals have been adopted within XMLSchema. Although several validators for XMLSchema appear to check for keys, relatively little attention has been paid to the general problem of how to check constraints in XML.

In this paper, we examine the problem of checking keys in XML documents and describe a native validator based on SAX. The algorithm relies on an indexing technique based on the paths found in key definitions, and can be used for checking the correctness of an entire document (bulk checking) as well as for checking updates as they are made to the document (incremental checking). The asymptotic performance of the algorithm is linear in the size of the document or update. We also discuss how XML keys can be checked in relational representations of XML documents, and compare the performance of our native validator against hand-coded relational constraints. Extrapolating from this experience, we propose how a relational schema can be designed to check XMLSchema key constraints using efficient relational PRIMARY KEY or UNIQUE constraints.

1 Introduction

Keys are an essential aspect of database design, and give the ability to identify a piece of data in an unambiguous way. They can be used to describe the correctness of data (constraints), to reference data (foreign keys), and to update data unambiguously.

The importance of keys for XML has recently been recognized, and several definitions have been introduced

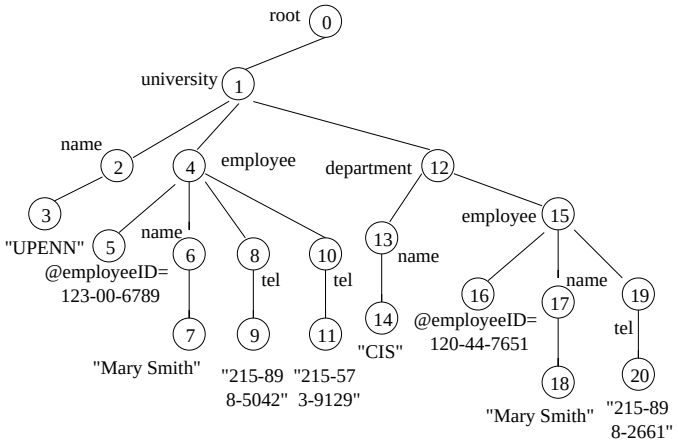

Figure 1: Tree representation of universities.xml

[1]. Aspects of these proposals have found their way into XMLSchema by the addition of UNIQUE and KEY constraints [2]. These proposals overcome a number of problems with the older notion of "ID" (and "IDREFS"): First, IDs are like oids and carry no real meaning in their value. In comparison, a key in the relational database sense is a set of attributes, and is value-based. Second, IDs must be globally unique. Third, they do not carry a notion of hierarchy, which is a distinguishing feature of XML.

As an example of the type of keys we might wish to define for an XML document, consider the sample document "universities.xml" represented in tree form in Figure 1. The document describes a set of universities, each of which has a set of departments. Employees can either work directly for the university or within a department. We might wish to state that the key of a university is its name. We might also wish to state that within a university an employee can be uniquely identified by her/his employeeID attribute (The symbol in Figure 1 denotes that employeeID is an attribute). Another key for an employee might be her/his telephone number (which can be a set of numbers) together with name, within the context of the whole repository.

Although definitions of keys for XML have been given, the question of how to best validate these constraints has not been solved. Building an efficient validator for key

constraints entails a number of challenges: First, unlike relational databases, keys are not localized but may be spread over a large part of the document. In our example, since university elements occur at a top level of nesting, the names of universities will be widely separated in the document. Second, keys can be defined within a particular context. In our example, employees are identified by their employeeID within the scope of a university. Third, an element may be keyed by more than one key constraint or may appear at different levels in the document (as with employees). Fourth, the validator should be incremental. That is, if an XML document has already been validated and an update occurs, it should be possible to validate just the update rather than the entire updated document, assuming that key information about the XML document is maintained.

The most straightforward strategy for building a validator is to develop a native XML key checker using SAX or DOM. Several XMLSchema validators have recently appeared which claim to support XMLSchema KEY and UNIQUE constraints [3, 4]. A native validator is also presented in this paper which differs from these validators by supporting a broader definition of XML keys than that given in [2] and by being incremental. The validator is based on a persistent key index and techniques to efficiently recognize the paths present in XML keys. Another approach for building a validator recognizes that the XML data may be stored in a relational database, and leverages relational technology of triggers and PRIMARY KEY/UNIQUE constraints to perform the checking.

To motivate the importance of the second strategy, consider a community of biomedial researchers who are performing gene expression experiments and, upon the recommendation of their bioinformatic experts, store their data directly in a relational database (see e.g. the Stanford Microarray Database [5], which uses Oracle). To exchange data, researchers convert their data into an agreed upon XML standard, MAGE-ML [6]. This standard includes a specification of keys, which are localized to each group (e.g. the context of keys for the standard is within a group which is identified by a given id). Each group is therefore expected to produce data that is correct with respect to the keys. Since the data is already stored in a relational database, it would be much more efficient to ensure that the data in relational form is correct with respect to the XML keys using relational technology than to produce the XML version of this data and then validate it before exporting. Or, if a group did not trust others to produce correct data, it would be more efficient to check the keys while inserting the imported XML data into their relational implementation.

How well the relational approach works depends strongly on how the data is stored. For example, suppose our sample data is stored using hybrid inlining [7]. Assuming the obvious DTD, this creates the following relational schema: *University(UID, Name), Department(DID, parentID, Name), Employee(EID, parentID, parentCODE, EmployeeID, Name), TelNumber(TID, tel, parentID).*

To enforce the first key (the name of a university is its key), we can specify *Name* to be PRIMARY KEY or UNIQUE for the *University* relation using SQL DDL. To enforce the second XML key (within a university, an employee can be uniquely identified by her/his employeeID), we must create a stored procedure which triggers upon update to join *Employee* with *University*, *Employee* with *Department* and *University*, and take the union of results. The checking procedure for the third key is even more complicated.

Moreover, suppose we have an XML file whose structure is extremely irregular, and therefore adopt an edge approach [7] for storage. In this case, checking even the simplest key constraint entails multiple joins and unions and will be very expensive.

The choice between a native strategy and a relational strategy is also influenced by the structure of the XML keys. In [1], the keys may be set-valued (*weak* keys) and may have a complex structure (i.e. the value of a key can be an XML sub-tree). In this case, validating key constraints using a relational database is extremely hard if not impossible. However, XMLSchema assumes that key values are either attributes or text and that they must occur exactly once (corresponding to a restriction of *strong* keys in [1]). In this case, we will show that it is possible to use relational technology, and advisable to do so if relational storage is already being used for storing the document.

The relational strategy also has several limitations: First, if the document is to be validated, the transformation to the relational schema must be information capacity preserving [8], at least with respect to the key information. This is not true for arbitrary transformations expressed, for example, in STORED [9]. Second, mapping XML key constraints to a fixed relational schema is an (as yet) unsolved problem (see [10] for preliminary results). However, if the schema can be modified then, for the restricted case of keys used in XMLSchema, this problem is solvable. Third, storing XML into an RDBMS involves a lot of overhead, and is not worth the cost unless the document will be used in that form (e.g. for efficient querying).

In this paper, we make the following contributions:

1. A native XML constraint validator, which can be used for XMLSchema KEY and UNIQUE constraints as well as for those in [1].

2. Bulk loading and incremental checking algorithms with complexity that is proportional to the size of the affected context (assuming a fixed number of keys are currently activate), hence is near optimal.

3. Experimental results showing the trade-off between our native approach and one based on relational technology.

4. Schema design techniques for validating XMLSchema keys using relational PRIMARY KEY/UNIQUE technology.

The rest of the paper is organized as follows: Section 2 introduces a definition of keys and presents our native XML constraint validator. Section 3 presents experimental results showing the trade-off between our native approach and one based on relational technology. Section 4 discusses schema design techniques for validating XMLSchema keys using relational PRIMARY KEY/UNIQUE technology, and discusses related work. We conclude with a summary and discussion of future work in Section 5.

2 XML Keys and the Native Validator

In defining a key for XML we specify three things: the context in which the key must hold, a set on which we are defining a key and the values which distinguish each element of the set. Since we are working with hierarchical data, specifying the context, set, and values involve path expressions.

Using the syntax of [1][1] a key can be written as

$$(Q, (Q', \{P_1, \ldots, P_p\}))$$

where Q, Q', and $P_1, \ldots, P_p$ are path expressions. Q is called the *context path*, Q' the *target path*, and $P_1, ..., P_p$ the *key paths*. The idea is that the context path Q identifies a set of *context nodes*; for each context node n, the key constraint must hold on the set of *target nodes* reachable from n via Q'.

For example, using XPath notation for paths, the keys of Section 1 can be written as:

$KS_1 = (/, (./university, \{./name\}))$: Within the context of the whole document ("/" denotes the empty path from the root), a university is identified by its name.

$KS_2 = (/university, (.//employee, \{./@employeeID\}))$: Within a university an employee can be uniquely identified by his/her employeeID (".//" refers to any sequence of labels).

$KS_3 = (/, (.//employee, \{./name, ./tel\}))$: Within the context of the whole document, an employee can be identified by their name and set of telephone numbers.

Definition 2.1: An XML tree T is said to *satisfy* a key iff for each context node n and for any target nodes m_1, m_2 reachable from n via Q', whenever there is a non-empty intersection of values for each key path $P_1, \ldots, P_p$ from m_1, m_2, then m_1 and m_2 must be the same node. ∎

For example, KS_3 is satisfied in the XML tree of Figure 1 since employee *123-00-6789* and *120-44-7651* are both within the context of the same university (*PENN*), and although they share the same name (*Mary Smith*) they do not share any telephone number. The key would also hold if we eliminated the telephone number for the first Mary Smith (*123-00-6789*) since $\emptyset \cap \{215-898-5042, 215-573-9129\} = \emptyset$. Although in these examples the key values are all sets of element of type string (text), in general the key values may be sets of XML trees. In this case, the notion of equality used to compute set intersection must be extended to one of tree equality.

While this definition of keys for XML is quite general, the one given in XMLSchema has the following restriction: Keys paths must be attributes or elements which occur exactly once and are text. This is a *strong key* defined in [1]. For example, KS_1 is only expressible if the *name* of a *university* is mandatory (and unique). Since our sample XML tree has multiple occurrences of *tel* within employees, KS_3 is not expressible. Furthermore, if the *name* of an *employee* had subelements *first name* and *last-name*, KS_2 would not be expressible since the key values are XML trees rather than text. Note that this definition of keys is tied to the schema while the definition of [1] does not require a schema. Also note that under the restrictions of XMLSchema, the key constraint states that target nodes must differ on some key value (analogous to the key constraint of relational databases). The XMLSchema UNIQUE constraint can also be captured by a key of form $(Q, (Q', \{\}))$ defined in [1] which state that, under a context node defined by Q, the target node defined by Q' is unique.

The path expression language used to define keys in XMLSchema is a restriction of XPath, and includes navigation along the child axis, disjunction at the top level, and wildcards in paths. This path language can be expressed as follows:

$$c ::= . \mid / \mid .q \mid /q \mid .//q \mid c|c$$

$$q ::= l \mid q/q \mid \_$$

Here "/" denotes the root or is used to concatenate two path expressions, "." denotes the current context, l is an element tag or attribute name, "_" matches a single label, and ".//" matches zero or more labels out of the root.

Note that just using key KS_2, we are not able to uniquely identify an *employee* node by its *employeeID*. That is, KS_2 is scoped within the context of a *university* node rather than within the scope of the root of the XML tree. However, given a key for the context node of KS_2, i.e. the *name* of a *university* (KS_1), we can then identify an *employee* node by its *employeeID*. The ability to recursively define context nodes up to the root of the tree is called a *transitive* set of keys [1]: $\{KS_1, KS_2\}$ is a transitive set of keys, as is $\{KS_3\}$ since its context is already the root of the tree. That is, KS_2 is scoped within the context of a university rather than within the scope of the root.

2.1 The XML Key Index

The XML constraint validator is based on a key index, which can be thought of in levels. The top level is the *key specification level*, which partitions the nodes in the XML tree according to their key specifications. Since a node may

[1] We adopt this because it is more concise than that of XMLSchema.

KS_1	0	name	UPENN	{1}
KS_2	1	employee	123-00-6789	{4}
		id	120-44-7651	{15}
KS_3	0	name	Mary Smith	{4,15}
		tel	215-898-5042	{4}
			215-573-9129	{4}
			215-898-2661	{15}

Figure 2: Key index for universities.xml

match more than one key specification, it may appear in more than one partition. The second level is the *context level*, which groups target nodes by their context. The third level is the *key path level*, which groups nodes based on key paths. The fourth level is the *key value level*, which groups target nodes by equivalence classes called key value sharing classes (*KVSC*). The KVSCs are defined such that the nodes in a class have some key nodes which are value-equivalent, following the same key path under the same context in a particular key. Since key values may be arbitrary XML trees, we store their serialized value (see [11] for details).

For example, the index structure for KS_1, KS_2 and KS_3 on the XML data in Figure 1 is shown in Figure 2. Note that nodes 4 and 15 are each keyed by KS_2 and KS_3, and that they share the same *name* value.

Given a new target node t within a context node c of an XML key K, the validator checks if t shares some key value with another target node under c for every key path. That is, for each key path P_i ($1 \leq i \leq p$) of K, it unions all the KVSCs that t belongs to and produces the set of nodes S_i that share some key value with t. It then computes $S = S_1 \cap ... \cap S_p$, which is the set of nodes that share some key value for all the key paths. If S contains more than one node (t), then K is violated.

For example, suppose we were validating the XML document of Figure 1 with respect to KS_1, KS_2 and KS_3. To check KS_3, as we parse through node 18 we find that the KVSC for *Mary Smith* is $\{4, 15\}$. As we continue the parse through node 19 we find that the KVSC for *215-898-2661* is $\{15\}$. Finishing the parse of the substructures of node 15, we check that $\{4, 15\} \cap \{15\} = \{15\}$, and so the constraint is valid.

Although the primary purpose of the index is to efficiently check keys, it can also be used to find a node using a transitive set of keys. This property will be used later when we talk about updating XML trees by specifying an update node.

Example 2.1: For example, suppose we want to find the *employee* whose *employeeID* is *120-44-7651* at *UPENN*. Since $\{KS_1, KS_2\}$ is a transitive set of keys, the query to locate the *employee* must specify a key for each context node. Here we use XQuery [12] for syntax.

```
<result>
{
for $a in document("universities.xml")/university
    $b in $a//employee
where boolean-and($a/name = "UPENN" ,
    $b/@employeeID = "120-44-7651" )
return $b
}
</result>
```

From the index of KS_1 in Figure 2, we know that *name* is a key of *university* and that the context is the root (node 0). The KVSC of *university* nodes with the key value *"UPENN"* following key path *name* is $\{1\}$. Since *@employeeID* is the key path of an *employee* node under the context of a *university* (KS_2), we can get the KVSC of *employee* nodes with the key value *"120-44-7651"* following the key path *@employeeID* under the context node 1. This class contains node 15. ■

2.2 Architecture for XML Constraint Validator

The architecture of our XML constraint validator is shown in Figure 3. The validator takes an XML key specification and document as input. Initially, the *start module* of the *Key manager* takes the key specification and sends the context path expression for each key to *DFA manager*. As the XML data streams into the SAX parser, events are generated and sent to DFAs in an *active DFA pool*; state transitions occur in response to these events. For each incoming path expression P, the *DFA manager* determines which DFA parses the path expression P, $DFA(P)$. If $DFA(P)$ is in the active DFA pool, the DFA manager modifies the current state set of $DFA(P)$ (to be described later); if $DFA(P)$ is in the *inactive DFA pool*, it will be activated and sent to the activate DFA pool; otherwise, if $DFA(P)$ is not in either pool, the DFA manager sends P to a *DFA constructor* which constructs $DFA(P)$ to parse P and put it into the active DFA pool. All DFAs in the active DFA pool make state transitions according to the event sent by the SAX parser. If any of them reaches its accepting state, it will signal the *PE(path expression) engine* of the Key Manager, which in turn decides the next path expression need to be parsed according to key specification. Any DFA that is not needed (to be described later) is deactivated and put in the inactive DFA pool.

2.3 Index Construction

As hinted at in the previous subsection, the index can be constructed in one pass over the XML file using a SAX parser and a set of DFAs which represent the context (Q), target (Q') and key paths ($P_1, \ldots, P_p$) for each XML key K. As the document streams in, each node is assigned a unique internal id. The internal id and tag of each node (the *node info*) is then communicated to the DFAs, which may trigger a state change.

Since a target node can only appear after its context node, $DFA(Q')$ is only activated when the accept state of $DFA(Q)$ has been reached. Similarly, $DFA(P_i), i = 1 \ldots p$ are only activated when the accept

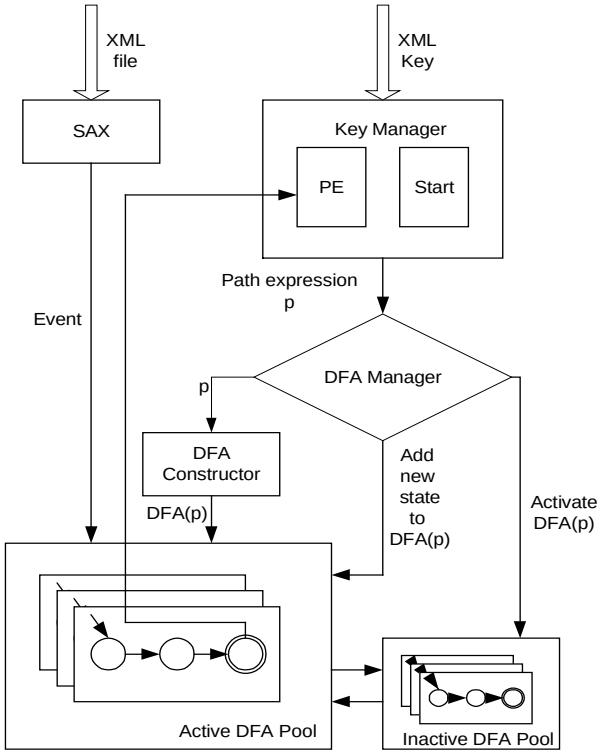

Figure 3: XML constraint validator architecture

state of $DFA(Q')$ has been reached. To keep track of which node infos are still current (i.e. tags whose corresponding closing tag has not yet been encountered), a stack is used.

When the accepting state for a context path is reached, the context node information is added to the index. Similarly, when the DFA for a key path reaches its final state, the id of the target node which activated the key path DFA and the key value recognized are added to the index. When the DFA for a target node reaches its final state, the process to check satisfaction of the key specification is invoked (discussed in Section 2.1).

Note that since the context and target path expressions may contain .//, several context nodes for one key and several target nodes for one context node can be activated at the same time. Our DFAs must therefore continue to seek the next match after reaching an accept state.

We optimize the algorithm in the following ways: First, the DFA manager constructs DFAs only when we try to match their path expression for the first time. Second, only one DFA is maintained per path expression, hence there is a set of *current states* representing all the activating nodes info. Each new tag that is encountered triggers the state transition for all current states, for each DFA in the active DFA pool. Third, we only activate a DFA when necessary. When a DFA is not being used, we deactivate it rather than destroy it. These optimizations allow us to construct at most once a single DFA for each path expression, and maintain it only for as long as necessary.

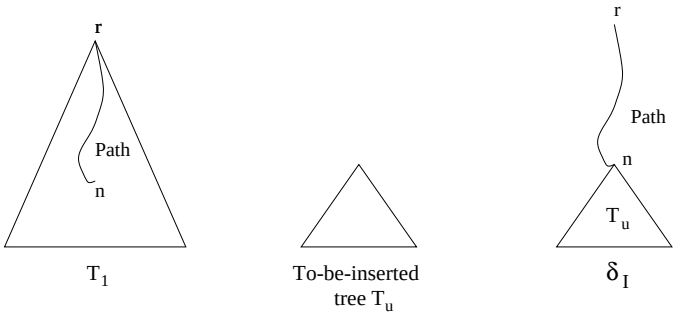

Figure 4: Delta XML tree for insertion

2.4 Incremental Maintenance

To describe how the native checker handles updates, we focus our attention on two basic (unordered) tree operations: insertion of a new tree below an update node, and deletion of the tree below an update node. These updates are specified as *insert*(n, T_u) and *delete*(n), where n is the internal id of the node to be updated, and T_u is a tree to be inserted.

For example, an update to *universities.xml* which gives the employee node 15 another telephone number "215-898-5042" could be written as $insert(15, T_u)$, where the content of T_u is `<tel> 215-898-5042 </tel>`. Node 15 could also be identified by a transitive set of key values as shown in Example 2.1.

Note that the XML standard for updates, XMLUpdate, has not yet been finalized, but currently includes many other operations, including specifying order in insertion, append, update and rename [13]. These operations could be handled within our framework, however limiting the updates considered simplifies the discussion. We can use any XML update language with (transitive) key values as predicates to locate the update node.

The incremental maintenance algorithm takes as input a delta XML tree, which reflects the changes to the initial XML document, and modifies the initial index so that is correct with respect to the updated XML tree.

A *delta XML tree* can be understood as follows: Given an update $insert(n, T_u)$, we create a tree T_n which is the path in the original document from the root to n. The delta XML tree δ_I is then generated by grafting T_u as a child of n in T_n (see Figure 4). Given an update $delete(n)$, the delta XML tree δ_D is formed by grafting the subtree rooted at n onto T_n.

Since our index is hierarchical, updates may affect the index at different levels. We can divide them into four cases by the effect of this update:

1. Entries at the context level are inserted or deleted.
   Note that bulk loading is a special case in which the delta XML tree is the entire tree.

2. One or more target nodes along with their key values are inserted or deleted.
   As an example, the insertion of an *employee* node as a child for the *university* node 1, that is, $insert(1, T_u)$ where the content of T_u is

`<employee><name>Judith Rodin</name> </employee>`
would be of this case for KS_2.

3. One or more key value(s) of an existing target node under some context are inserted or deleted.
   The effect on KS_3 for the update $insert(15, T_u)$, where the content of T_u is
   `<tel> 215-898-5042 </tel>`,
   is an example of this case.

4. The key value is changed.
   This case can only happen when the key value is a tree instead of a text node and we are inserting or deleting a subtree of a key node under an existing target node. For example, consider a modified version of the tree in Figure 1 in which the *name* of *employee* node has two children: *first name* and *last name* (e.g. node 6 with label *name* has a *first name* node with value *Mary* and a *last name* node with value *Smith*). If we delete the *first name*, then the key value of the *employee* node 4 in KS_3 will be changed and key constraints need to checked.

It is clear that since insertions introduce new values, the index must be maintained whenever the insertion interacts with the context, target or key path of some key. Deletion is more surprising: Although deletions in relational databases can never violate a key constraint, in the context of XML they may change some key value. Therefore the index must be maintained whenever a deletion interacts with some key path (case 4 above). The next question will be how to determine when an update "interacts" with a context, target or key path expression. This can be done by reasoning about the concatenation of labels from the root to the update node in T_n, and the paths Q, $Q.Q'$ and $Q.Q'.P_j$. Details can be found in the technical report [11].

In the last two cases, a new key value for a node is inserted into the index. It turns out that it is quite inefficient to check if this causes a key constraint violation using only the key index presented so far since it entails retrieving all the key values of the updated node. We therefore build an auxiliary index on the key index to retrieve these key values efficiently, which indexes each target nodes under their context node. For each key path P_j, it keeps a pointer to the key values for the target node.

Example 2.2: Consider the insertion of a telephone number
`<tel> 215-898-5042 </tel>`
to the *employee* with *employeeID =120-44-7651* within the *university* whose *name= UPENN*.
From Example 2.1, we can find the id of the update node (15) and construct the delta XML tree. It is easy to see that this update does not affect key specifications KS_1 and KS_2. It does, however, affect KS_3 by inserting a new key value (case 3). Processing the delta XML tree will result in the updated index structure of Figure 5. Following the pointers for node 15 in the auxiliary index structure, we

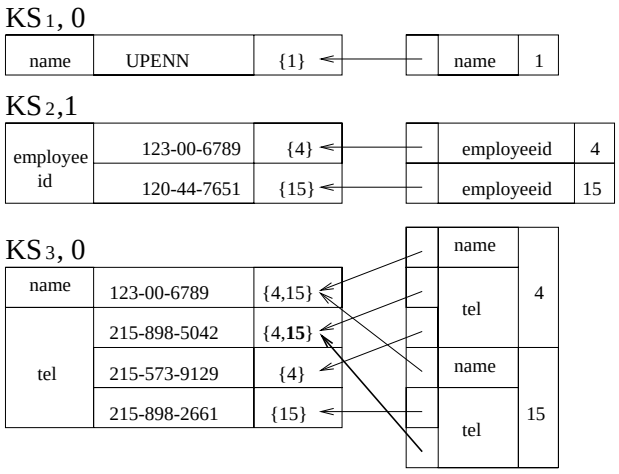

Figure 5: updated key index example

can find all the KVSCs it belongs to: the KVSC for *Mary Smith* ($\{4, 15\}$), the KVSC for *215-898-2661* ($\{15\}$), and the KVSC for *215-898-5042* ($\{4, 15\}$). To check KS_3, we union the two KVSCs for key path *tel* and get a set $\{4, 15\}$. When we intersect this with the KVSC for key path *name* we get a conflicting node set, $\{4, 15\}$. Since a violation is discovered, the update is rolled back. ■

Theorem 2.3: The asymptotic performance of the bulk loading and incremental algorithms is linear in the size of the affected context of the document, assuming that there are only a constant number of active states in each key at any give time.

Proof: According to the assumption, the number of DFAs and the number of DFA active states are constant. Therefore, when the SAX parser sends an event to the active DFAs the total number of state changes is constant. On the other hand, the number of events is proportional to the size of the affected context of the document. So we can say that the asymptotic performance of the algorithms is linear in the size of the affected context of the document. ■

This assumption appears to hold true in practice, as will be seen in the experimental results in Section 3. We have studied several real data sets and found that the distribution of the data is quite uniform. This means that at any given time, in practice the number of active DFAs and the number of active states of each DFA is bounded by a constant.

The details and analysis of these algorithms can be found in [11].

3 Experimental Results

To compare the performance of our native key validator versus using a relational approach, we store an XML document in a commercial relational database system using hybrid inlining and handcode the key constraints.[2] All experiments run on the same 1.5GHz Pentium 4 machine with

[2]We omit experiments using shared inlining because hybrid inlining offers better performance.

```
<!ELEMENT db(university*)>
<!ELEMENT university(name,school*,department*,
                     employee*)>
<!ELEMENT school(name, department*,employee*)>
<!ELEMENT department(name, researchgroup*,
                     employee*)>
<!ELEMENT researchgroup(name, employee*)>
<!ELEMENT employee(name, employeeID)>
<!ELEMENT name(#PCDATA)>
```

Figure 6: DTD of universities.xml

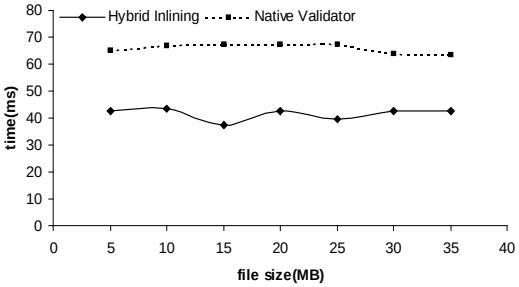

Figure 7: Time to incrementally check KS_4

512MB memory and one hard disk with 7200rpm. The operating system is Windows 2000, and the DBMS is DB2 universal version 7.2 using the high-performance storage option. We use Java 2 to code the program and JDBC to connect to the database.

We do not report results for the edge mapping approach. Since it is based on the structure of XML rather than the semantics of the document, checking even the simplest of keys (e.g. the first key below) is several order of magnitude slower than the native validator.

3.1 Data set and keys

We use a synthetic data set generated by an XML Generator from the XML Benchmark project [14]. (We also ran experiments on real data sets, EMBL [15]. Since the results were similar, we omit them.) XML Generator was modified to generate a series of XML files of different sizes, according to DTD shown in Figure 6. Using hybrid-inlining [7], we create the following relational tables:
University(uID, name),
School(sID, name, parentID),
Department(dID, name, parentID, parentCode),
ResearchGroup(rID, name, parentID), and
Employee(eID, name, employeeID, parentID, parentCode).

The keys to be validated are similar to those used earlier:

$KS_4 := (/, (./university, \{./name\}))$: Each *university* is identified by its *name.*

$KS_5 := (/university, (.//department, \{./name\}))$: Within a *university*, each *department* is identified by its *name.*

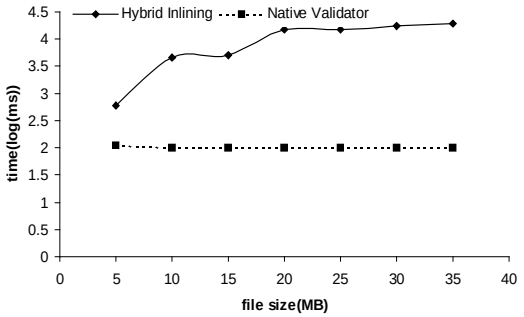

Figure 8: Time to incrementally check KS_5

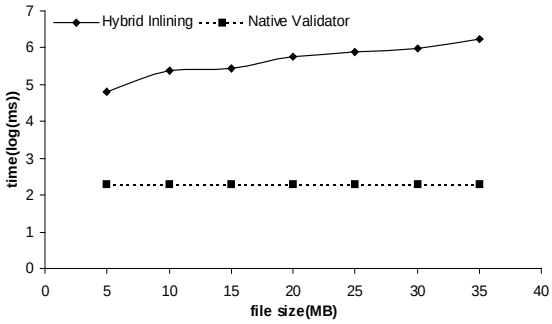

Figure 9: Time to incrementally check KS_6

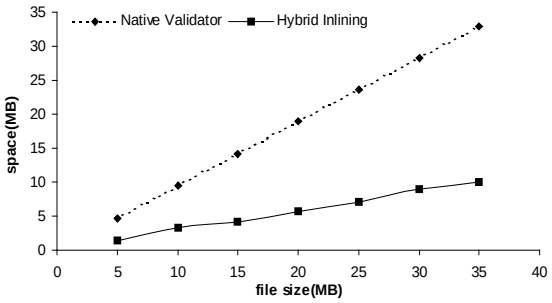

Figure 10: Index size for KS_6

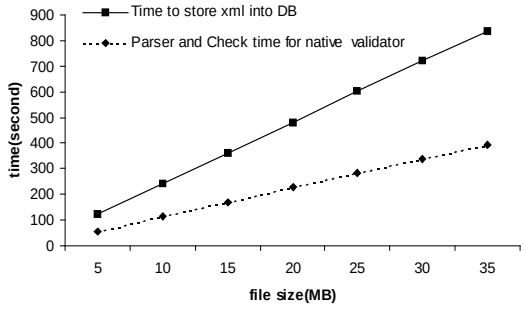

Figure 11: Time to store XML document in RDBMS vs. native validator

KS_6: $= (/university, (.//employee, \{./employeeID, \}))$: Within a *university*, at whatever level they occur, each *employee* is uniquely identified by his/her *employeeID*.

To check KS_4, we specify attribute *name* to be the key for the *university* table. To check KS_5, we need to join *school* with *department* whose parent is *school* and union it with the *department* whose parent is *university* to get all possible *(university.uid, department.name)* pairs, and then check if there are any duplicates. Checking KS_6 is similar to KS_5 except more joins and unions are needed to get *(university.uid, employeeID)* pairs. Indices on *(parentCode, parentID)* or *(parentID)* are built on every table where applicable. To speed up key checking, we also build index (*name*, *parentCode*, *parentID*) on *Department*, and (*employeeID*, *parentCode*, *parentID*) on *Employee*.

3.2 Experiments

We model incremental updates by inserting a delta XML document of size 100KB into XML documents of different sizes. We plot the time needed for the relational approach versus the native validator time over a series of files of increasing sizes in Figure 7, 8, 9. Checks of KS_4,KS_5, and KS_6 are performed independently. Since the native validator is much faster than relational checking for KS_5, and KS_6, we use log scale for the Y axis. Note that the native validator is only slightly slower than using PRIMARY KEY (KS_4), and that its time is roughly constant since the update size is constant.

A comparison of the native validator key index size versus that of relational indices specifically designed for checking KS_6 is shown in Figure 10; results for KS_5 are similar. Our index is somewhat larger than the relational indices, however, the native validator is not currently optimized for space.

Figure 11 illustrates that unless an RDBMS is already being used as the storage strategy for the XML document, it should not be used just to check keys: the time needed to store the XML document is much larger than just using the native validator.

4 Discussion and Related Work

We now restrict our attention to keys as defined as in XMLSchema: the key value for each key path must exist and be unique, and each key path is a text or attribute node. It turns out that for this form of keys, it is possible to design relational schemas to perform efficient key checks.

The results of the previous section show that if the XML document is stored using relational technology then the fastest way to check an XML key is to use PRIMARY KEY or UNIQUE constraints. Therefore, it is important to design the relational schema with the keys in mind.

Example 4.1: Suppose we generate the relational instance shown in Figure 12 using hybrid inlining for the data of Figure 1 in which all telephone numbers are eliminated.

University

uid	Name
200	UPENN

Department

did	parID	Name
300	200	CIS

Employee

eid	parID	CODE	EmpID	Name
100	200	univ	...6789	Mary Smith
101	300	dept	...7651	Mary Smith

Figure 12: Relational instance using inlining

KS_2 can only be checked by a query involving joins and unions over this relational design. However, if we add the following relation:

KS2

uid	eid	EmpID
200	100	...6789
200	101	...7651

then KS_2 can be checked by stating that (uid, EmpID) is PRIMARY KEY. To eliminate redundancy, we could have merged this table with Employee to create Employee(eid, parID, parCODE, uID, EmpID, Name), where (eid) is PRIMARY KEY, and (uid, EmpID) (for KS_2) is UNIQUE. ■

Generalizing from this example, what we need is a relational structure which mirrors the key structure. That is, for each XML key $(C, (T, \{K_1, ..., K_p))$, it is sufficient to ensure that a table T containing an attribute c representing the id of the context node, an attribute t representing the id of a target node, and attributes $k_i (i = 1..p)$ for each key path is present in the relational schema. Note that c and t represent the relational database's internal id for the context and target elements. We can then define a PRIMARY KEY or UNIQUE constraint on $(C, K_1, ..., K_p)$ for T. Note that no matter what XML to relational mapping strategy is used – even the edge approach, which does not require a DTD – this redundant table can always be built.

We must also ensure that the mapping from the XML instance to these key checking tables is complete. That is, the populated table must contain every match for C, T and $K_1, ..., K_p$ in the document to ensure that whenever there is an XML key violation in the XML document it is caught by the relational key constraint check. Updates to the database must be made exclusively through the XML interface, and each one must be handled as a transaction.

The correctness of this approach follows from the correctness of the algorithm in Section 2 [11].

In some ways, this is analogous to designing relational schemas in 3NF, where a minimal basis is computed for a given set of functional dependencies and a schema is output corresponding to this basis [16]. Computing the minimal basis relies on a sound and complete set of inference rules for functional dependencies (Armstrong's Axioms).

Unfortunately, little is known about computing a minimal basis of XMLSchema keys. Using a restricted path language, in [17] we have given a sound and complete set of inference rules for keys as defined in Section 2. The inference problem for XML keys is complicated by the fact that it involves reasoning about inclusion of path ex-

pressions. Since the restricted version of XPath used in XMLSchema is not comparable to that of [17], these rules must be rethought before they can be used to compute a minimal basis for XMLSchema keys.

Fortunately, the question of minimality of the XML keys is orthogonal to the question of ensuring that whenever there is an XML key violation in the XML document it is caught by the relational key constraint check.

Related Work. There are several native XMLSchema checkers and validators: XML-Schema-Quality-Checker of IBM [18] takes as input an XMLSchema and diagnoses improper uses of the schema language. However, it is not a validating parser, that is, it does not take as input an instance document and validate it against the schema. Microsoft XML Parser 4.0(MSXML)[3] is a validating parser, but does not currently support regular expressions and appears to have some bugs with respect to keys. The University of Edinburgh has an on-going schema validator project called XSV, but does not appear to have implemented XMLSchema keys [4].

The salient differences between the approach taken in these XMLSchema key validators and the one suggested in this paper are as follows. First, our definition of XML keys follows that of [1] which is more general than that given in XMLSchema. However, our key checker can easily be used to validate XMLSchema keys. Second, we have designed an incremental validation algorithm which verifies updates to an XML document. Other approaches are designed to parse the entire updated XML file to check the key constraints.

[19] proposed *a lazy DFA* where a DFA processing a set of path expressions is constructed from the NFA at runtime. This technique can also be used for DFA optimization in our native validator.

There are many proposals for mapping XML into relational databases. The edge approach described in [20] maps each edge in the XML tree to a tuple in a relation, thus capturing the structure rather than the semantics of XML data. The inlining techniques of [7] store XML into a relational database based on a DTD. They do not consider keys in this mapping, and in fact there may be conflicts between constraints expressed in a DTD and those expressed as keys. For example, the DTD `<ELEMENT foo (X,X)>` and the key $(\emptyset, (X, \{\}))$ contradict each other [1]. LegoDB [21] is a cost-based XML to relational mapping, and explores alternatives based inlining/outlining and union factorization/distribution to favor a given query workload. However, this approach does not consider keys and may not guarantee the completeness of the transformation with respect to the keys. The Clio system [22] preserves certain constraints when performing the schema mapping, but loses keys. The XML-relational constraint mapping scheme mentioned at the beginning of this section is therefore (to our knowledge) the first XML storage mapping technique that preserves XML keys.

Note that another approach for mapping keys is to express them as XQuery queries and automatically translate them into SQL. For example, the key KS_5 can be expressed as the following XQuery:

```
for $c in Document("universities.xml")/university,
    $t1 in $c//department,
    $t2 in $c//department
where boolean-and(not(node-equal($t1,$t2)),
       $t1/name$t2/name)
return $t1, $t2
```

If the result is the empty set, then the constraint is valid with respect to the data. Given the relational schema using hybrid inlining in the experiment and using the automatic mapping suggested in the XPERANTO project [23, 24], the corresponding SQL would be:

```
select did
from (select uid,dn, did, count(*) as c
      from (select department.name dn, did, uid
            from department, university
            where ParentCode = "university"
                  and ParentID = uid
            union
            select department.name dn,did,uid
            from school, department, university
            where department.Pcode = "school"
                  and department.ParentID = sid
                  and school.ParentID = uid
            ) as tmp
      group by uid, dn) as tmp2
where c>1;
```

Such SQL queries are inefficient compared to using PRIMARY KEY/UNIQUE constraints. Using a constraint preserving mapping with key tables is therefore a much better approach.

5 Conclusions

In this paper, we focused on the problem of validating key constraints over XML documents. We discussed two alternative approaches: One approach is to validate XML key constraints using a native key checker. Although native validators for XMLSchema have been proposed, few have considered KEY and UNIQUE constraints. Our native XML constraint validator differs from these approaches in that it considers a broader class of keys than defined in XMLSchema, in which the value of keys may be XML trees rather than simple text and key paths can be set valued. The validator can be used for both bulk-loading (i.e. one pass over the entire document) and incremental checking (i.e. XML updates to the document can be processed and checked against a persistent key index for the file). Our validator can also be used with a little modification to check referential integrity in XMLSchema (KEYREF), since it already provides the ability to find a node according to its key value.

The other approach is to leverage relational technology. Observing that stored procedures involving joins are much more expensive to evaluate than PRIMARY KEY/UNIQUE constraints, we proposed designing the relational schema to include relations which mirror the XML keys. When these key relations are populated in a way that preserves all key information in the original XML document, XML keys can be efficiently checked using PRIMARY KEY/UNIQUE constraints. This approach will work for KEY and UNIQUE constraints as defined in XMLSchema, or more generally, for strong XML keys as defined in [1]. It does not work for weak XML keys [1]. To our knowledge, this is the first XML-to-relational schema mapping that considers key constraints.

Experiments showing the trade-off between our XML key validator and relational techniques were also performed. The experiments show that the performance of our native validator for XML keys is roughly the same as PRIMARY KEY/UNIQUE checks in a relational database. However, our native validator performs better by several orders of magnitude than when the key checks are performed using complex stored procedures. It is therefore important to carefully design the relational schema if frequent updates are expected to take advantage of PRIMARY KEY/UNIQUE checks. Since the cost (time and space) to store an XML document in a relational database is high compared to the time and space of a native validator, however, the relational approach should only be used if the document is being stored relationally for other reasons (such as optimizing queries).

At the heart of both our native and relational approaches is the data structure introduced in Section 2 called the key index. Compared with other XML index structures [25, 26, 27, 28], the index captures both the structure and the content information of the data. A query evaluator can therefore use this index together with information about path restriction and value conditions to optimize queries on keys. The preliminary results shows that our index gives better performance than that of [28] for key look-ups in XML. Similar as the approach of key look-ups, our validator can efficiently enforce the foreign-key constraint(the XMLSchema countpart is keyref) in bulk loading as well as incremental maintenance.

In future work we plan to explore its use for more general queries. For example, for high frequency queries we could build a set of indexes which match the queries and can be used to efficiently retrieve the query result. For lower frequency queries, we can see if the key and high frequency query indexes match a portion of the query.

References

[1] P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. Keys for XML. In *WWW10*, 2001.

[2] H. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML Schema Part 0: Primer , May 2001. http://www.w3.org/TR/xmlschema-0/.

[3] Microsoft XML Parser 4.0(MSXML). Available at: http://msdn.microsoft.com.

[4] XML Schema Validator. Available at: http://www.ltg.ed.ac.uk/ ht/xsv-status.html.

[5] G. Sherlock and et al. T. Hernandez-Boussard. The stanford microarray database. In *Nucleic Acids Research, 29(1):152–155, 2001.*, 2001. http://daisy.stanford.edu/MicroArray/SMD.

[6] MAGE-ML. http://www.mged.org/Workgroups/MAGE/mage-ml.html.

[7] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *The VLDB Journal*, pages 302–314, 1999.

[8] R. J. Miller, Y. E. Ioannidis, and R Ramakrishnan. The use of information capacity in schema integration and translation. In *Proc. 19th International VLDB Conference*, pages 120–133, August 1993.

[9] Alin Deutsch, Mary Fernandez, and Dan Suciu. Storing semistructured data with STORED. In *In Proceedings of the Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats*, pages 431–442, 1999.

[10] Susan Davidson Carmem Hara Wenfei Fan. Propagating xml keys to relations. Technical Report MS-CIS-01-31, University of Pennsylvania, Computer and Information Science Department, 2001.

[11] Y.Chen, S. Davidson, and Y. Zheng. Indexing keys in hierarchical data. Technical Report MS-CIS-01-30, University of Pennsylvania, Computer and Information Science Department, 2001.

[12] XQuery 1.0: An XML query language, June 2001. http://www.w3.org/XML/Query.

[13] XUpdate. http://www.xmldb.org/xupdate/xupdate-wd.html.

[14] XMARK the XML-benchmark project, April 2001. http://monetdb.cwi.nl/xml/index.html.

[15] D. G. Higgins, R. Fuchs, P. J. Stoehr, and G. N. Cameron. The EMBL data library. *Nucleic Acids Research*, 20:2071–2074, 1992.

[16] Jeffrey D. Ullman. *Principles of Database and Knowledgebase Systems I*. Computer Science Press, Rockville, MD 20850, 1989.

[17] P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. Reasoning about keys for XML. In *International Workshop on Database Programming Languages (DBPL)*, 2001.

[18] XML
Schema Quality Checker, February 2002. Available at: http://www.alphaworks.ibm.com/tech/xmlsqc.

[19] T.J.Green, M. Onizuka, and D. Suciu. Processing xml streams with deterministic automata and stream indexes, 2001. unpublished.

[20] D. Florescu and D. Kossmann. Storing and querying XML data using an RDBMS. In *Bulletin of the Technical Committee on Data Engineering*, pages 27–34, September 1999.

[21] Phil Bohannon, Juliana Freire, Prasan Roy, and Jerome Simeon. From XML-Schema to Relations: A Cost-Based Approach to XML Storage. In *ICDE*, 2002.

[22] Ling-Ling Yan, Renee J. Miller, Laura M. Haas, and Ronald Fagin. Data-driven understanding and refinement of schema mappings. In *SIGMOD Conference*, 2001.

[23] Michael J. Carey, Daniela Florescu, Zachary G. Ives, Ying Lu, Jayavel Shanmugasundaram, Eugene J. Shekita, and Subbu N. Subramanian. XPERANTO: Publishing Object-Relational Data as XML. In *WebDB (Informal Proceedings)*, pages 105–110, 2000.

[24] J. Shanmugasundaram, J. Kiernan, E. Shekita, C. Fan, and J. Funderburk. Querying XML Views of Relational Data. In *Proceedings of the 21th International Conference on VLDB*, 2001.

[25] J. McHugh, J. Widom, S. Abiteboul, Q. Luo, and A. Rajaraman. Indexing semistructured data. Technical report, Stanford University, Computer Science Department, 1998.

[26] T. Milo and D. Suciu. Index structures for path expressions. In *ICDT*, 1999.

[27] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for efficient indexing of paths in graph structured data. In *ICDE*, 2002.

[28] Quanzhong Li and Bongki Moon. Indexing and querying XML data for regular path expressions. In *The VLDB Journal*, pages 361–370, 2001.